20. Debugging, Exceptions, and Error Handling

Created: April 1, 2003
Updated: September 16, 2003

Debugging

The overview for this chapter consists of the following topics:

- Introduction
- Chapter Outline

Introduction

This chapter discuss the debugging mechanisms available in the NCBI C++ toolkit. There are two approaches to getting more information about an application, which does not behave correctly:

- Investigate the application's log without recompiling the program,
- Add more diagnostics and recompile the program.

Of course, there is always the third method which is to run the program under an external debugger. While using an external debugger is a viable option, this method relies on an external program and not on a log or diagnostics produced by the program itself which in many cases is customized to reflect the program behavior, and can, therefore, more quickly reveal the source of errors.

Chapter Outline

The following is an outline of the topics presented in this chapter:

- Extracting Debug Data
    - Command Line Parameters.
    - Getting More Trace Data.
        - Tracing
        - Diagnostic Messages
    - Tracing in the Connection Library

Extracting Debug Data

The C++ Toolkit has several mechanisms which can be used by a programmer to extract information about the program usage, printing trace and diagnostic messages, and examining the object state dump. The following sections discuss these topics in more detail:

- Command Line Parameters.

- Getting More Trace Data.
- Tracing in the Connection Library
- NCBI C++ Toolkit Diagnostics
- Object state dump
- Exceptions

Command Line Parameters.

There are several command line parameters (see Table 1), which are applicable to any program which utilizes NCBI C++ toolkit, namely CNcbiApplication class. They provide with the possibility

- to obtain a general description of the program as well as description of all available command line parameters (*-h* flag),
- to redirect the program's diagnostic messages into a specified file (*-logfile* key),
- to read the program's configuration data from a specified file (*-conffile* key).

Table 1. Command line parameters available for use to any program that uses CNcbiApplication

Flag	Description	Example
-h	Print description of the application's command line parameters.	`theapp -h`
-logfile	Redirect program's log into the specified file	`theapp -logfile theapp_log`
-conffile	Read the program's configuration data from the specified file	`theapp -conffile theapp_cfg`

Getting More Trace Data.

All NCBI C++ toolkit libraries produce a good deal of diagnostic messages. Still, many of them remain "invisible" - as long as the tracing is disabled. Some tracing data is only available in debug builds - see `_TRACE` macro for example. Other - e.g., the one produced by `ERR_POST` or `LOG_POST` macros - could be disabled. There are three ways to manipulate these settings, that is enable or disable tracing, or set the severity level of messages to print:

- from the application itself,
- from the application's configuration file,

- with the help of environment variables.

The following additional topics relating to trace data are presented in the subsections that follow:

- Tracing
- Diagnostic Messages

Tracing

There are two ways to post trace messages: using _TRACE or ERR_POST macro Trace messages produced with the help of _TRACE macro are only available in debug mode, while those posted by ERR_POST are available in both release and debug builds. By default, tracing is disabled. See Table 2 for settings to enable tracing.

Table 2. Enabling Tracing

C++ toolkit API	Configuration file	Environment
call `SetDiagTrace (eDT_Enable);`	define *DIAG_TRACE* entry in the *DEBUG* section: `[DEBUG] DIAG_TRACE=1`	define `DIAG_TRACE` environment variable: `set DIAG_TRACE=1`

Please note, when enabling trace from a configuration file, some trace messages could be lost: before configuration file is found and read the application may assume that the trace was disabled. The only way to enable tracing from the very beginning is by setting the environment variable.

Diagnostic Messages

Diagnostic messages produced by ERR_POST macro are available both in debug and release builds. Such messages have a severity level, which defines whether the message will be actually printed or not, and whether the program will be aborted or not. To change the severity level threshold for posting diagnostic messages, see Table 3.

Table 3. Changing severity level for diagnostic messages

C++ toolkit API	Configuration file	Environment
call `SetDiagPostLevel (EDiagSev postSev);` Valid arguments are `eDiag_Info`, `eDiag_Warning`,	define *DIAG_POST_LEVEL* entry in the *DEBUG* section: `[DEBUG] DIAG_POST_LEVEL=Info` Valid values are *Info*, *Warning*, *Error*, *Critical*, *Fatal*.	define `DIAG_POST_LEVEL` environment variable: `set DIAG_POST_LEVEL=Info` Valid values are *Info*, *Warning*, *Error*, *Critical*, *Fatal*.

C++ toolkit API	Configuration file	Environment
`eDiag_Error`, `eDiag_Critical`, `eDiag_Fatal`.		

Only those messages, which severity is equal or exceeds the threshold will be posted. By default, messages posted with *Fatal* severity level also abort execution of the program. This can be changed by ***SetDiagDieLevel(EDiagSev dieSev)*** API function.

Tracing in the Connection Library

The connection library has its own tracing options. It is possible to print the connection parameters each time the link is established, and even log all data transmitted through the socket during the life of the connection (see Table 4.

Table 4. Setting up trace options for connection library

	Configuration file	Environment
Connection parameters:	define *DEBUG_PRINTOUT* entry in the *CONN* section: `[CONN] DEBUG_PRINTOUT=TRUE` Valid values are *TRUE*, or *YES*, or *SOME*.	define `CONN_DEBUG_PRINTOUT` environment variable: `set CONN_DEBUG_PRINTOUT=TRUE` Valid values are *TRUE*, or *YES*, or *SOME*.
All data:	define *DEBUG_PRINTOUT* entry in the *CONN* section: `[CONN] DEBUG_PRINTOUT=ALL` Valid values are *ALL*, or *DATA*.	define `CONN_DEBUG_PRINTOUT` environment variable: `set CONN_DEBUG_PRINTOUT=ALL`Valid values are *ALL*, or *DATA*.

NCBI C++ Toolkit Diagnostics

NCBI C++ toolkit provides with a sophisticated diagnostic mechanism. Diagnostic messages could be redirected to different output channels. It is possible to set up what additional information should be printed with a message, for example date/time stamp, file name, line number etc. Some macros are defined only in debug mode:`_TRACE`, `_ASSERT`, `_TROUBLE`. Others are also defined in release mode as well: `_VERIFY`, `THROW*_TRACE`.

Object state dump

Potentially useful technique in case of trouble is to use object state dump API. In order to use it, the object's class must be derived from CDebugDumpable class, and implementation of the class should supply meaningful dump data in its ***DebugDump*** function. Debug dump gives an object's state snapshot, which can help in identifying the cause of problem at run time.

Exceptions

NCBI C++ toolkit defines its own type of C++ exceptions. Unlike standard ones, this class

- makes it possible to define error codes (specific to each exception class), which could be analyzed from a program,
- provides with more information about where a particular exception has been thrown from (file name and line number),
- gives the possibility to create a stack of exceptions to accumulate a backlog of events (unfinished jobs) which caused the problem,
- has elaborated, customizable reporting mechanism,
- supports using standard diagnostic mechanism with all the configuration options it provides.

NCBI C++ Error Handling and Diagnostics

The following topics are discussed in this section:

- Debug-mode for Internal Use
- C++ Exceptions
- Standard NCBI C++ Message Posting

Debug-mode for Internal Use

`#include <corelib/ncbidbg.hpp>` [also included in <corelib/ncbistd.hpp>]

There are four preprocessor macros (`_TROUBLE`, `_ASSERT`, `_VERIFY` and `_TRACE`) to help the developer to catch some (logical) errors on the early stages of code development and to hardcode some assertions on the code and data behaviour for internal use. All these macros gets disabled in the non-debug versions lest to affect the application performance and functionality; to turn them on, one must *#define* the `_DEBUG` preprocessor variable. Developer must be careful and do not use any code with side effects in `_ASSERT` or `_TRACE` as this will cause a discrepancy in functionality between debug and non-debug code. For example, `_ASSERT(a++)` and *_TRACE ("a++ = " << a++)* would increment `"a"` in the debug version but do nothing in the non-debug one).

- `_TROUBLE` -- Has absolutely no effect if `_DEBUG` is not defined; otherwise, unconditionally halt the application.
- `_ASSERT(expr)` -- Has absolutely no effect if `_DEBUG` is not defined; otherwise, evaluate expression `expr` and halt the application if `expr` resulted in zero(or *"false"*).

- `_VERIFY(expr)` -- Evaluate expression `expr`; if `_DEBUG` is defined and `expr` resulted in zero(or *"false"*) then halt the application.

- `_TRACE(message)` -- Has absolutely no effect if `_DEBUG` is not defined; otherwise, it outputs the `message` using Standard NCBI C++ message posting. NOTE: as a matter of fact, the tracing is turned off by default, even if `_DEBUG` is defined, and you still have to do a special configuration to really turn it on.

All these macros automatically report the file name and line number to the diagnostics. For example, this code located in file *"somefile.cpp"* at line 333:

```
int x = 100;
_TRACE( "x + 5 = " << (x + 5) );
```

will output:

```
"somefile.cpp", line 333: Trace: x + 5 = 105
```

C++ Exceptions

`#include <corelib/ncbiexpt.hpp>` [also included in <corelib/ncbistd.hpp>]

The following additional topics are discussed in this section:

- Standard C++ Exception Classes, and Two Useful NCBI Exception Classes (CErrnoTemplException, CParseTemplException)
- Using STD_CATCH*(...) to catch and report exceptions
- Using THROW*_TRACE(...) to throw exceptions
- THROWS*(...) -- Exception Specification

Standard C++ Exception Classes, and Two Useful NCBI Exception Classes (CErrnoTemplException, CParseTemplException)

One must use standard C++ exceptions as much as possible. There is also a couple of auxiliary exception classes derived from ***std::runtime_error***:

- CErrnoException -- to report failure in a standard C library function; it automatically appends to the user message a system-specific description reported by `errno`

- CParseException -- to report an erroneous position (passed in the additional constructor parameter) along with the user message

Then, it is **strictly recommended** that when the basic functionality provided by standard C++ exceptions is unsufficient for some reason, one must derive the new ad hoc exception classes from one of the standard exception classes. -- This is to provide a more uniform way of exceprion handling, for we could smartly catch/handle most of thrown exceptions using `STD_CATCH(message)` and `STD_CATCH_ALL(message)` preprocessor macros.

Using `STD_CATCH*(...)` to *catch* and report exceptions

You can use `STD_CATCH(message)` macro to catch an exception derived from the standard exception class (***std::exception***) -- when all you want to do about this exception is just to print out the "message" along with the info passed with the ***std::exception::what()***. `STD_CATCH_ALL(message)` first tries to catch a ***std::exception***-derived exception (with `STD_CATCH(message)`); and if the thrown exception is not "standard" then it posts the "message".

The "message" argument can be of any form acceptable by the diagnostic class ***CNcbiDiag***. This way, the easy way of dealing with exception in the NCBI C++ code will be like:

```
class foreign_exception { ..... };
class exception_derived_user : public exception { ..... };
........... char arg1 = "qqq";
int arg2 = 888;
try {
    SomeFunc(arg1, arg2);
} catch (foreign_exception& fe) {
    // do something special with the particular "non-standard"
    // (not derived from "std::exception") exception "foreign_exception"
} catch (exception_derived_user& eu) {
    // do something special with the particular "standard"
    // (derived from "std::exception") exception "exception_derived_user"
}
// handle all the rest "standard" exceptions in a uniform way
STD_CATCH ( "in SomeFunc(" << arg1 << "," << arg2 << ")" );
```

Here, if ***SomeFunc*** do *throw std::runtime_error("Invalid Arg2");* then the application will print out (to its diagnostic stream) something like:

```
Error: [in SomeFunc(qqq,888)] Exception: Invalid Arg2
```

Using `THROW*_TRACE(...)` to *throw* exceptions

If you use one of `THROW*_TRACE(...)` macros to *throw* an exception, and the source was compiled in a debug mode (i.e. with the preprocessor `_DEBUG` defined), then you can turn on the following features that proved to be very useful for debugging:

- If the tracing is on, then the location of the *throw* in the source code and the thrown exception will be printed out to the current diagnostic stream, e.g.:

  ```
  THROW_TRACE(CParseException, ("Failed parsing(at pos. 123)", 123));
  ```

  ```
  "coretest.cpp", line 708: Trace: CParseException: {123}
  ```

```
Failed parsing(at pos. 123)

--------------------------------

strtod("1e-999999", 0);
THROW1_TRACE(CErrnoException, "Failed strtod('1e-999999', 0)");

"coretest.cpp", line 718: Trace: CErrnoException:
Failed strtod('1e-999999', 0): Result too large
```

- Sometimes, it can be convenient to just abort the program execution at the place where you throw an exception, e.g. in order to examine the program stack and overall state that led to this *throw*. By default, this feature is not activated. You can turn it on for your whole application by either setting the environment variable `$ABORT_ON_THROW` to an arbitrary non-empty string, or by setting the application's registry entry *ABORT_ON_THROW* (in the *[DEBUG]* section) to an arbitrary non-empty value. You also can turn it on and off in your program code, calling function SetThrowTraceAbort().

NOTE: if the source was not compiled in the debug mode, then the `THROW*_TRACE(...)` would just *throw* the specified exception, without doing any of the "fancy stuff" we just described.

`THROWS*(...)` -- Exception Specification

One is encouraged to write exception specifications for NCBI C++ functions. However, due to some discrepancy in how different compilers handle *unexpected* exception events we decided to use `THROWS_NONE` and `THROWS()` preprocessor macros for the case of "poor" compilers.

Thus, you must use:

```
void f1(int i) THROWS_NONE;
int  f2(void) THROWS((e0));
int  f3(long) THROWS((e1,e2));
```

in the place of:

```
void f1(int i) throw();
int  f2(void) throw(e0);
int  f3(long) throw(e1,e2);
```

respectively. -- Please note the double parenthesis for `THROWS()`.

Standard NCBI C++ Message Posting

`#include <corelib/ncbidiag.hpp>` [also included in <corelib/ncbistd.hpp>]

Standard diagnostics is provided with the ***CNcbiDiag*** class. A given application can have as many objects of this class as needed. An important point to remember is that each instance of the ***CNcbiDiag*** class actually stores (and allows to append to) only one message at a time. When the message controlled by an instance of ***CNcbiDiag*** is complete, ***CNcbiDiag*** invokes the ***Post()*** method of a global handler object (of type ***CDiagHandler***) and passes the message (along with its severity level) as the method's argument.

Usually, this global object would merely dump the message to a diagnostic stream, and there is an auxiliary function ***SetDiagStream()*** that can be used to specify the output stream for the diagnostics. One can call ***SetDiagStream(&NcbiCerr)*** to dump the diagnostics to the standard error output stream:

```
/// Set diagnostic stream.
///
/// Error diagnostics are written to output stream "os".
/// This uses the SetDiagHandler() functionality.
NCBI_XNCBI_EXPORT
extern void SetDiagStream
(CNcbiOstream* os,
bool          quick_flush  = true,///< Do stream flush after every message
FDiagCleanup  cleanup      = 0,   ///< Call "cleanup(cleanup_data)" if diag.
void*         cleanup_data = 0    ///< Stream is changed (see SetDiagHandler)
);
```

Using ***SetDiagHandler()***, one can install a custom handler object of type ***CDiagHandler*** to process the messages posted via ***CNcbiDiag***. The implementation of the ***CStreamDiagHandler*** in *"ncbidiag.cpp"* is a good example of how to do this.

```
/////////////////////////////////////////////////////////////////////////////
///
/// CDiagHandler --
///
/// Base diagnostic handler class.

class NCBI_XNCBI_EXPORT CDiagHandler
{
public:
    /// Destructor.
    virtual ~CDiagHandler(void) {}

    /// Post message to handler.
    virtual void Post(const SDiagMessage& mess) = 0;
};

/// Set the diagnostic handler using the specified diagnostic handler class.
NCBI_XNCBI_EXPORT
extern void SetDiagHandler(CDiagHandler* handler,
                           bool can_delete = true);

/// Get the currently set diagnostic handler class.
NCBI_XNCBI_EXPORT
extern CDiagHandler* GetDiagHandler(bool take_ownership = false);
```

where:

```
/////////////////////////////////////////////////////////////////////////////
///
/// SDiagMessage --
///
/// Diagnostic message structure.
///
/// Defines structure of the "data" message that is used with message handler
/// function("func"),  and destructor("cleanup").
/// The "func(..., data)" to be called when any instance of "CNcbiDiagBuffer"
/// has a new diagnostic message completed and ready to post.
/// "cleanup(data)" will be called whenever this hook gets replaced and
/// on the program termination.
/// NOTE 1:  "func()", "cleanup()" and "g_SetDiagHandler()" calls are
///          MT-protected, so that they would never be called simultaneously
///          from different threads.
/// NOTE 2:  By default, the errors will be written to standard error stream.

struct SDiagMessage {
    /// Initalize SDiagMessage fields.
    SDiagMessage(EDiagSev severity, const char* buf, size_t len,
                 const char* file = 0, size_t line = 0,
                 TDiagPostFlags flags = eDPF_Default, const char* prefix = 0,
                 int err_code = 0, int err_subcode = 0,
                 const char* err_text = 0);

    mutable EDiagSev m_Severity;   ///< Severity level
    const char*      m_Buffer;     ///< Not guaranteed to be '\0'-terminated!
    size_t           m_BufferLen;  ///< Length of m_Buffer
    const char*      m_File;       ///< File name
    size_t           m_Line;       ///< Line number in file
    int              m_ErrCode;    ///< Error code
    int              m_ErrSubCode; ///< Sub Error code
    TDiagPostFlags   m_Flags;      ///< Bitwise OR of "EDiagPostFlag"
    const char*      m_Prefix;     ///< Prefix string
    const char*      m_ErrText;    ///< Sometimes 'error' has no numeric code,
                                   ///< but can be represented as text

    // Compose a message string in the standard format(see also "flags"):
    //    "<file>", line <line>: <severity>: [<prefix>] <message> [EOL]
    // and put it to string "str", or write to an output stream "os".

    /// Which write flags should be output in diagnostic message.
    enum EDiagWriteFlags {
        fNone   = 0x0,      ///< No flags
        fNoEndl = 0x01      ///< No end of line
    };

    typedef int TDiagWriteFlags; /// Binary OR of "EDiagWriteFlags"

    /// Write to string.
    void Write(string& str, TDiagWriteFlags flags = fNone) const;
```

```
    /// Write to stream.
    CNcbiOstream& Write(CNcbiOstream& os, TDiagWriteFlags flags = fNone) const;
};
```

Installing a new handler typically destroys the previous handler, which can be a problem if you need to keep the old handler around for some reason. There are two ways to address this issue:

- Declare an object of class ***CDiagRestorer*** at the top of the block of code in which you will be using your new handler. This will protect the old handler from destruction, and automatically restore it -- along with any other diagnostic settings -- when the block exits in any fashion. As such, you can safely use the result of calling ***GetDiagHandler()*** at the beginning of the block even if you have changed the handler within the block.
- Call ***GetDiagHandler(true)*** and then destroy the old handler yourself when done with it. This works in some circumstances in which ***CDiagRestorer*** is unsuitable, but places much more responsibility on your code.

For compatibility with older code, the diagnostic system also supports specifying simple callbacks:

```
/// Diagnostic handler function type.
typedef void (*FDiagHandler)(const SDiagMessage& mess);

/// Diagnostic cleanup function type.
typedef void (*FDiagCleanup)(void* data);

/// Set the diagnostic handler using the specified diagnostic handler class.
NCBI_XNCBI_EXPORT
extern void SetDiagHandler(CDiagHandler* handler,
                           bool can_delete = true);
```

However, it is better to use the object-based interface for new code.

The following additional topics are discussed in this section:

- Formatting and Manipulators
- ERR_POST macro
- Turn on the Tracing

Formatting and Manipulators

To compose a diagnostic message with ***CNcbiDiag*** you can use the formatting operator "<<". It works practically the same way as operator "<<" for standard C++ output streams. ***CNcbiDiag*** class also has some ***CNcbiDiag***-specific *manipulators* to control the message severity level:

- ***Info*** -- set severity level to `eDiag_Info`
- ***Warning*** -- set severity level to `eDiag_Warning`
- ***Error*** -- set severity level to `eDiag_Error` [default]
- ***Fatal*** -- set severity level to `eDiag_Fatal`
- ***Trace*** -- set severity level to `eDiag_Trace`

NOTE: whenever the severity level is changed, ***CNcbiDiag*** also automatically executes the following two *manipulators*:

- ***Endm*** -- means that the message is complete and to be flushed(via the global callback as described above)
- ***Reset*** -- directs to discard the content of presently composed message

The ***Endm*** manipulator also gets executed on the ***CNcbiDiag*** object destruction.

For example, this code:

```
int iii = 1234;
CNcbiDiag diag1;

diag1 << "Message1_Start " << iii;
         // message 1 is started but not ready yet
{ CNcbiDiag diag2; diag2 << Info << "Message2"; }
         // message 2 flushed in destructor
diag1 << "Message1_End" << Endm;
         // message 1 finished and flushed by "Endm"
diag1 << "Message1_1"; // will be flushed by the following "Warning"
diag1 << Warning << "Discard this warning" << ++iii << Reset;
         // message discarded
diag1 << "This is a warning " << iii;
diag1 << Endm;
```

will write to the diagnostic stream(if the latter was set with ***SetDiagStream()***):

```
Error: Message1_Start 1234
Info: Message2
Error: Message1_End
Error: Message1_1
Warning: This is a warning 1235
```

ERR_POST macro

There is an `ERR_POST(message)` macro that can be used to shorten the error posting code. This macro is discussed in the chapter on Core Library.

Turn on the Tracing

The tracing (messages with severity level `eDiag_Trace`) is considered to be a special, debug-oriented feature, and therefore it is not affected by ***SetDiagPostLevel()*** and ***SetDiagDieLevel()***. To turn the tracing on or off in your code you can use function SetDiagTrace().

By default, the tracing is off -- unless you assign environment variable `$DIAG_TRACE` to an arbitrary non-empty string (or, alternatively, you can set *DIAG_TRACE* entry in the *[DEBUG]* section of your registry to any non-empty value).

DebugDump: Take an Object State Snapshot

The following topics are discussed in this section:

- Terminology
- Requirements
- Architecture
- Implementation
- Examples

Debugging is an inevitable part of software developement. When it comes to a "mystical" problem, one can spend days and days hunting for a glitch. So, being prepared is not just a "nice thing to have", it is a requirement.

When a system being developed crashes consistently, debugging is easy in the sense that the problem is reproducable. Were that all bugs like this! It is much more "fun", when the system crashes intermittently, under circumstances about which we have only a vague idea, if any, of the symptoms or the cause. What the developer needs in this case is information - the more the better. One short message ("Assertion failed") is good and a coredump is better, but we typically need a more user-friendly reporting of the program status at the point of failure.

One possible idea is to make the object tell about itself. That is, in case of trouble (but not necessarily trouble), an object could call a function that would report as much as possible about itself and other object it contains or to which it refers. During such operation the object should not do anything important, something that could potentially cause other problems. The diagnostic must of course be safe - it should only take a snapshot of an object's state and never alter that data.

Sure, ***DebugDump*** may cause problems by itself, even if everything is "correct". Let us say there are two objects, which "know" each other: `Object A` refers to `Object B`, while `Object B` refers to `Object A` (very common scenario in fact). Now dumping contents of `Object A` will cause dumping of `Object B`, which in turn will cause dumping of `Object A`, and so on until the stack overflows.

Terminology

So, dumping the object contents should look as a single function call, i.e. something like this:

```
Object name;
...
name.DebugDump(?);
```

The packet of information produced by such operation we call *bundle*. The class ***Object*** is most likely derived from other classes. The function should be called sequentially for each subclass, so it could print its data members. The piece of information produced by the subclass we call *frame*. The object may refer to other objects. Dumping of such object produces a *sub-bundle*, which consists of its own *frames*. To help fight cyclicity, we introduce *depth* of the dump. When an object being dumped wants to dump other objects it refers to, it should reduce the *depth* by one. If the *depth* is already zero, other objects should not be dumped.

Requirements

- The dump data should be separated from its representation. That is, the object should only supply data, something else should format it. Examples of formatting may include generating human-readable text or file in a special format (HTML, XML), or even transmitting the data over the network.
- Debug and release libraries should be compatible.
- It should be globally configurable as to whether the dump produces any output or not,

Architecture

Class ***CDebugDumpable*** is a special abstract base class. Its purpose is to define a virtual function ***DebugDump***, which any derived class should implement. Another purpose is to store any global dump options. Any real dump should be initiated through a non-virtual function of this class - so, global option could be applied. Class ***CObject*** is derived from this class. So, any classes based on ***CObject*** may benefit from this functionality right away. Other classes may use this class as a base later on (e.g. using multiple inheritance).

Class ***CDebugDumpContext*** provides a generic dump interface for dumpable objects. The class has nothing to do with data representation. Its purpose is the ability to describe the location of where the data comes from, accept it from the object and transfer to the data formatter.

Class ***CDebugDumpFormatter*** defines the dump formatting interface. It is an abstract class.

Class ***CDebugDumpFormatterText*** is derived from ***CDebugDumpFormatter***. Based on incoming data, it generates a human-readable text and passes it into any output stream (***ostream***).

In general, the system works like this:

1. Client creates DebugDump formatter object (it could be an object of class ***CDebugDumpFormatterText*** or any other class derived from ***CDebugDumpFormatter***) and passes it to a proper, non-virtual function of the object to be dumped. Bundle name is to be defined here - it can be anything, but a reasonable guess would be to specify the location of the call and the name of the object being dumped.

2. ***CDebugDumpable*** analyses global settings, creates ***CDebugDumpContext*** object and calls virtual DebugDump() function of the object.

3. DebugDump function of each subclass defines a frame name (which must be the type of the subclass), calls DebugDump function of a base class and finally logs its own data members. From within the DebugDump(), the object being dumped "sees" only ***CDebugDumpContext***. It does not know any specifics about target format in which dump data will be eventually represented.

Implementation

The following topics are discussed in this section:

- CDebugDumpable
- CDebugDumpContext
- CDebugDumpFormatter

CDebugDumpable

The class is an abstract one. Global options are stored as static variable(s).

```
public:
    // Enable/disable debug dump
    static void EnableDebugDump(bool on);

    // Dump using text formatter
    void DebugDumpText(ostream& out,
                       const string& bundle, unsigned int depth) const;
    // Dump using external dump formatter
    void DebugDumpFormat(CDebugDumpFormatter& ddf,
                   const string& bundle, unsigned int depth) const;

    // Function that does the dump - to be overloaded
    virtual void DebugDump(CDebugDumpContext ddc, unsigned int depth) const = 0;
```

Any derived class must impelement a relevant DebugDump function.

CDebugDumpContext

The class defines a public dump interface for a client object. It receives the data from the object and decides when and what functions of dump formatter to call.

The dump interface looks like this:

```
public:
    CDebugDumpContext(CDebugDumpFormatter& formatter, const string& bundle);
    // This is not exactly a copy constructor -
    // this mechanism is used internally to find out
    // where are we on the Dump tree
    CDebugDumpContext(CDebugDumpContext& ddc);
    CDebugDumpContext(CDebugDumpContext& ddc, const string& bundle);

public:
    // First thing in DebugDump() function - call this function
    // providing class type as the frame name
    void SetFrame(const string& frame);
    // Log data in the form [name, data, comment]
    // All data is passed to a formatter as string, still sometimes
    // it is probably worth to emphasize that the data is REALLY a string
    void Log(const string& name, const string& value, bool is_string = true, const string&
comment = kEmptyStr);
    void Log(const string& name, bool value,          const string& comment = kEmptyStr);
    void Log(const string& name, long value,          const string& comment = kEmptyStr);
    void Log(const string& name, unsigned long value, const string& comment = kEmptyStr);
    void Log(const string& name, double value,        const string& comment = kEmptyStr);
    void Log(const string& name, const void* value,   const string& comment = kEmptyStr);
    void Log(const string& name, const CDebugDumpable* value, unsigned int depth);
```

A number of overloaded ***Log*** functions is provided for convenience only.

CDebugDumpFormatter

This abstract class defines dump formatting interface:

```
public:
    virtual bool StartBundle(unsigned int level, const string& bundle) = 0;
    virtual void EndBundle(  unsigned int level, const string& bundle) = 0;

    virtual bool StartFrame( unsigned int level, const string& frame) = 0;
    virtual void EndFrame(   unsigned int level, const string& frame) = 0;

    virtual void PutValue(   unsigned int level, const string& name,
                             const string& value, bool is_string,
                             const string& comment) = 0;
```

Examples

Supposed that there is an object `m_ccObj` of class ***CSomeObject*** derived from ***CObject***. In order to dump it into the standard `cerr` stream, one should do one of the following:

```
m_ccObj.DebugDumpText(cerr, "m_ccObj", 0);
```

or

```
{
    CDebugDumpFormatterText ddf(cerr);
    m_ccObj.DebugDumpFormat(ddf, "m_ccObj", 0);
}
```

The ***DebugDump*** function should look like this:

```
void CSomeObject::DebugDump(CDebugDumpContext ddc, unsigned int depth) const
{
    ddc.SetFrame("CSomeObject");
    CObject::DebugDump(ddc,depth);
    ddc.Log("m_1", m_1);
    ddc.Log("m_2", m_2);
    ... etc for each data member
}
```

Exception Handling (%20) in the NCBI C++ Toolkit

The following topics are discussed in this section:

- NCBI C++ Exceptions
- The CErrnoTemplException Class
- The CParseTemplException Class
- Macros for Standard C++ Exception Handling
- Exception Tracing

NCBI C++ Exceptions

C++ exceptions is a standard mechanism of communicating abnormal or unexpected events to a higher execution context. By throwing an exception a piece of code says it was unable to complete the task and it is up to others to decide what to do next.

What the standard mechanism lacks is backlog, history of unfinished tasks and its consequences. Say for instance, a program tries to load some data from a database. An exception occurs, which says a connection to some port could not be created -- so what? How meaningfull is it? What did the program try to do? Where did the request for the connection come from?

Another problem is analyzing and handling exceptions in a program. When an exception is caught, what is known for sure is only that something bad has happened -- but what exactly? The standard exception has only type (exception class) and a text message. The latter probably makes sense for a human, but not for a program. The former does not seem to be clear enough.

The following topics are discussed in this section:

- Requirements
- Architecture

- Implementation
- Examples

Requirements

In order for exceptions to be more useful, they should meet the following requirements:

- Exceptions should contain information about where exactly has it been thrown -- for a human.
- Exceptions should have a numeric id -- for a program.
- It should be possible to create a stack of exceptions -- to accumulate a backlog of events (unfinished jobs) which caused the problem. Still, for a client, it should look like a single exception. That is, a client should be able to ignore completely the compound structure of the exception being thrown and still get some meaningful information.
- The system should provide for the ability to analyze the exception backlog and possibly print information about each exception separately.
- It should be possible to report the exception data into an arbitrary output channel and possibly format it differently for each channel.

Architecture

Each subsystem (library) has its own type of exceptions. It may have several types, if necessary, but all of them should be derived from a single base class (which in turn is derived from a system-wide base class). So, the type of an exception uniquely identifies the library which produced it.

Each exception has a numeric id, which is unique throughout the subsystem. Such an id gives an unambiguous description of the problem occurred. Each id is associated with a text message. Strictly speaking, there is only one message associated with a given id, so there is no need to include the message in the exception itself -- it could be taken from an external source. Still, we suggest using the message -- it serves as an additional comment. Also, it does not restrict us from using an external source of messages in the future.

Each exception has information about the location where it has been thrown -- file name and line number.

An exception can have a reference to the "lower level" one, which makes it possible to analyze the backlog. Naturally, such a backlog cannot be created automatically - it is a developer's responsibility. The system only provides the mechanism, it does not solve problems by itself. The developer is supposed to catch exceptions in proper places and re-throw them with the backlog information added.

The exception constructor's mandatory parameters include location information, exception id and a message. This constructor is to be used at the lower level, when the exception is thrown initially. At higher levels we need a constructor, which would accept the exception from the lower level as one of its parameters.

The NCBI exception mechanism has a sophisticated reporting mechanism -- the standard ***exception::what()*** function is definitely not enough. There are three groups of reporting mechanisms:

- exception formats its data by itself and either returns the result as a string or puts it into an output stream;
- client provides an external exception data formatter;
- NCBI standard diagnostic mechanism is used.

Implementation

The following topics are discussed in this section:

- CException
- Derived exceptions
- Reporting an exception
- CExceptionReporter
- Choosing and analyzing error codes

CException

There is a single system-wide exception base class -- ***CException***. Each subsystem **must** implement its own type of exceptions, which must be be derived from this class. The class defines basic requirements of an exception construction, backlog and reporting mechanisms.

The ***CException*** constructor includes location information, exception id and a message. Each exception class defines its own error codes. So, the error code "by itself" is meaningless -- one should also know the the exception class, which produced it.

```
/// Constructor.
///
/// When throwing an exception initially, "prev_exception" must be 0.
CException(const char* file, int line,
                       const CException* prev_exception,
                       EErrCode err_code,const string& message) throw();
```

To make it easier to throw/re-throw an exception, the following macros are defined:

```
NCBI_THROW(exception_class, err_code, message)
NCBI_RETHROW(prev_exception, exception_class, err_code,message)
NCBI_RETHROW_SAME(prev_exception, message)
```

The last one (`NCBI_RETHROW_SAME`) re-throws the same exception with backlog information added.

The ***CException*** class has numerous reporting methods (the contents of reports is defined by diagnostics post flags):

```
/// Standard report (includes full backlog).
virtual const char* what(void) const throw();

/// Report the exception.
///
/// Report the exception using "reporter" exception reporter.
/// If "reporter" is not specified (value 0), then use the default
/// reporter as set with CExceptionReporter::SetDefault.
void Report(const char* file, int line,
            const string& title, CExceptionReporter* reporter = 0,
            TDiagPostFlags flags = eDPF_Trace) const;

/// Report this exception only.
///
/// Report as a string this exception only. No backlog is attached.
string ReportThis(TDiagPostFlags flags = eDPF_Trace) const;

/// Report all exceptions.
///
/// Report as a string all exceptions. Include full backlog.
string ReportAll (TDiagPostFlags flags = eDPF_Trace) const;

/// Report "standard" attributes.
///
/// Report "standard" attributes (file, line, type, err.code, user message)
/// into the "out" stream (this exception only, no backlog).
void ReportStd(ostream& out, TDiagPostFlags flags = eDPF_Trace) const;

/// Report "non-standard" attributes.
///
/// Report "non-standard" attributes (those of derived class) into the
/// "out" stream.
virtual void ReportExtra(ostream& out) const;

/// Enable background reporting.
///
/// If background reporting is enabled, then calling what() or ReportAll()
/// would also report exception to the default exception reporter.
/// @return
///   The previous state of the flag.
static bool EnableBackgroundReporting(bool enable);
```

Also, the following macro is defined that calls the ***CExceptionReporter::ReportDefault()*** method to produce a report for the exception:

```
NCBI_REPORT_EXCEPTION(title,e)
```

Finally, the following data access functions help to analyze exception from a program:

```
    /// Get class name as a string.
    virtual const char* GetType(void) const;

    /// Get error code interpreted as text.
    virtual const char* GetErrCodeString(void) const;

    /// Get file name used for reporting.
    const string& GetFile(void) const;

    /// Get line number where error occurred.
    int GetLine(void) const;

    /// Get error code.
    EErrCode GetErrCode(void) const;

    /// Get message string.
    const string& GetMsg (void) const;

    /// Get "previous" exception from the backlog.
    const CException* GetPredecessor(void) const;
```

Derived exceptions

The only requirement for a derived exception is to define error codes as well as its textual representation. Implementation of several other functions (e.g. constructors) are, in general case, pretty straightforward -- so we put it into a macro definition, `NCBI_EXCEPTION_DEFAULT`. Please note, this macro can only be used when the derived class has no additional data members. Here is an example of an exception declaration:

```
class CSubsystemException : public CException
{
public:
    /// Error types that subsystem can generate.
    enum EErrCode {
        eType1,        ///< Meaning of eType1
        eType2         ///< Meaning of eType2
    };

    /// Translate from the error code value to its string representation.
    virtual const char* GetErrCodeString(void) const
    {
        switch (GetErrCode()) {
        case eType1: return "eType1";
        case eType2: return "eType2";
        default:     return CException::GetErrCodeString();
        }
    }

    // Standard exception boilerplate code.
    NCBI_EXCEPTION_DEFAULT(CSubsystemException, CException);
};
```

In case the derived exception has data members not found in the base class, it should also implement its own ***ReportExtra*** method -- to report this non-standard data.

Reporting an exception

There are several way to report an NCBI C++ exception:

1. An exception is capable of formatting its own data, returning a string (or a pointer to a string buffer). Each exception report occupies one line. Still, since an exception may contain a backlog of previously thrown exceptions, the resulting report could contain several lines of text - one for each exception thrown. The report normally contains information about the location from which the exception has been thrown, the text representation of the exception class and error code, and a description of the error. The content of the report is defined by diagnostics post flags. The following methods generate reports of this type:

```
/// Standard report (includes full backlog).
virtual const char* what(void) const throw();

/// Report the exception.
///
/// Report the exception using "reporter" exception reporter.
/// If "reporter" is not specified (value 0), then use the default
/// reporter as set with CExceptionReporter::SetDefault.
void Report(const char* file, int line,
            const string& title, CExceptionReporter* reporter = 0,
            TDiagPostFlags flags = eDPF_Trace) const;

/// Report this exception only.
///
/// Report as a string this exception only. No backlog is attached.
string ReportThis(TDiagPostFlags flags = eDPF_Trace) const;

/// Report all exceptions.
///
/// Report as a string all exceptions. Include full backlog.
string ReportAll (TDiagPostFlags flags = eDPF_Trace) const;

/// Report "standard" attributes.
///
/// Report "standard" attributes (file, line, type, err.code, user message)
/// into the "out" stream (this exception only, no backlog).
void ReportStd(ostream& out, TDiagPostFlags flags = eDPF_Trace) const;
```

Functions ***what()*** and ***ReportAll()*** may also generate a *background* report - the one generated by a default exception reporter. This feature can be disabled by calling the static method

```
CException::EnableBackgroundReporting(false);
```

2. A client can provide its own exception reporter. An object of this class may either use exception data access functions to create its own reports, or redirect reports into its own output channel(s). While it is possible to specify the reporter in the ***CException::Report()*** function, it is better if the same reporting functions are used for exceptions, to install the reporter as a default one instead, using

```
CExceptionReporter::SetDefault(const CExceptionReporter* handler);
```

static function, and use the standard `NCBI_REPORT_EXCEPTION` macro in the program.

3. Still another way to report an exception is to use the standard diagnostic mechanism provided by NCBI C++ toolkit. In this case the code to generate the report would look like this:

```
try {
    ...
} catch (CException& e) {
    ERR_POST("your message here" << e);
}
```

CExceptionReporter

One of possible ways to report an exception is to use an external "reporter" modeled by the ***CExceptionReporter*** abstract class. The reporter is an object that formats exception data and sends it to its own output channel. A client can install its own, custom exception reporter. This is not required, though. In case the default was not set, the standard NCBI diagnostic mechanism is used.

The ***CExceptionReporter*** is an abstract class, which defines the reporter interface:

```
/// Set default reporter.
static void SetDefault(const CExceptionReporter* handler);

/// Get default reporter.
static const CExceptionReporter* GetDefault(void);

/// Enable/disable using default reporter.
///
/// @return
///   Previous state of this flag.
static bool EnableDefault(bool enable);

/// Report exception using default reporter.
static void ReportDefault(const char* file, int line,
                          const string& title, const CException& ex,
                          TDiagPostFlags flags = eDPF_Trace);
```

```
    /// Report exception with _this_ reporter
    virtual void Report(const char* file, int line,
                        const string& title, const CException& ex,
                        TDiagPostFlags flags = eDPF_Trace) const = 0;
```

Choosing and analyzing error codes

Choosing and interpreting error codes can potentially create some problems because each exception class has its own error codes, and interpretation. Error codes are implemented as an enum type, ***EErrCode***, and the enumerated values are stored internally in a program as numbers. So, the same number can be interpreted incorrectly for a different exception class than the one in which the enum type was defined. Say for instance, there is an exception class, which is derived from ***CSubsystemException*** -- let us call it ***CBiggersystemException*** -- which also defines two error codes: `eBigger1` and `eBigger2`:

```
class CBiggersystemException : public CSubsystemException
{
public:
    /// Error types that subsystem can generate.
    enum EErrCode {
        eBigger1,        ///< Meaning of error code, eBigger1
        eBigger2         ///< Meaning of error code, eBigger2
    };

    /// Translate from the error code value to its string representation.
    virtual const char* GetErrCodeString(void) const
    {
        switch (GetErrCode()) {
        case eBigger1: return "eBigger1";
        case eBigger2: return "eBigger2";
        default:      return CException::GetErrCodeString();
        }
    }

    // Standard exception boilerplate code.
    NCBI_EXCEPTION_DEFAULT(CBiggersystemException, CSubsystemException);
};
```

Now, suppose an exception ***CBiggersystemException*** has been thrown somewhere. On a higher level it has been caught as ***CSubsystemException***. It is easy to see that the error code returned by the ***CSubsystemException*** object would be completely meaningless: the error code of ***CBiggersystemException*** cannot be interpreted in terms of ***CSubsystemException***.

One reasonable solution seems to be isolating error codes of different exception classes -- by assigning different numeric values to them. And this has to be done by the developer. Such isolation should only be done within each branch of derivatives only. Another solution is to make sure that the exception in question does belong to the desired class, not to any intermediate classes in the derivation hierarchy. The template function ***UppermostCast()*** can be used to perform this check:

```
/// Return valid pointer to uppermost derived class only if "from" is _really_
/// the object of the desired type.
///
/// Do not cast to intermediate types (return NULL if such cast is attempted).
template <class TTo, class TFrom>
const TTo* UppermostCast(const TFrom& from)
{
    return typeid(from) == typeid(TTo) ? dynamic_cast<const TTo*>(&from) : 0;
}
```

UppermostCast() utilizes the runtime information using the ***typeid()*** function, and dynamic cast conversion to return either a pointer to "uppermost" exception object or NULL.

The following shows how ***UppermostCast()*** can be used to catch the correct error types:

```
try {
    ...
    NCBI_THROW(CBiggersystemException,eBigger1,"your message here");
    ...
}
catch (CSubsystemException& e) {
    // call to UppermostCast<CSubsystemException>(e) would return 0 here!
    // which means that "e" was actually the object of a different class
    const CBiggersystemException *p = UppermostCast<CBiggersystemException>(e);
    if (p) {
        switch (p->GetErrCode()) {
        case CBiggersystemException::eBigger1:
            ...
            break;
        case CBiggersystemException::eBigger2:
            ...
            break;
        default:
            ...
            break;
        }
    }
    NCBI_RETHROW_SAME(e,"your message here");
}
```

It is possible to use the runtime information to do it even better. Since **GetErrCode** function is non-virtual, it might check the type of the object, for which it has been called, against the type of the class to which it belong. If these two do not match, the function returns *invalid* error code. Such code only means that the caller did not know the correct type of the exception, and the function is unable to interpret it.

Examples

The following topics are discussed in this section:

- Throwing an exception
- Reporting an exception

Throwing an exception

It is important to remember that the system only provides a mechanism to create a backlog of unfinished tasks, it does not create this backlog automatically. It is up to developer to catch exceptions and re-throw them with the backlog information added. Here is an example of throwing ***CSubsystemException*** exception:

```
... // your code
NCBI_THROW(CSubsystemException,eType1,"your message here");
...
```

The code that catches, and possibly re-throws the exception might look like this:

```
try {
    ... // your code
} catch (CSubsystemException& e) {
    if (e.GetErrCode() == CSubsystemException::eType2) {
        ...
    } else {
        NCBI_RETHROW(e, CSubsystemException, eType1, " your message here")
    }
} catch (CException& e) {
    NCBI_RETHROW(e, CException, eUnknown, "your message here")
}
```

Reporting an exception

There are a number of ways to report ***CException***, for example:

```
try {
    ... // your code
} catch (CSubsystemException& e) {
    NCBI_REPORT_EXCEPTION("your message here", e);
    ERR_POST(e);
    cerr << e.ReportAll();
    cerr << e.what();
    e.Report(__FILE__, __LINE__, "your message here");
}
```

We suggest using `NCBI_REPORT_EXCEPTION(title,e)` macro (which is equivalent to calling *e.Report(__FILE__,__LINE__,title)*) - it redirects the output into standard diagnostic channels and is highly configurable.

The CErrnoTemplException Class

The ***CErrnoTemplException*** class is a template class used for generating error exception classes:

```
/////////////////////////////////////////////////////////////////////////////
///
/// CErrnoTemplException --
///
/// Define template class for easy generation of Errno-like exception classes.

template<class TBase> class CErrnoTemplException :
                        public CErrnoTemplExceptionEx<TBase, CStrErrAdapt::strerror>
{
public:
    /// Parent class type.
    typedef CErrnoTemplExceptionEx<TBase, CStrErrAdapt::strerror> CParent;

    /// Constructor.
    CErrnoTemplException<TBase>(const char* file,int line,
        const CException* prev_exception,
        typename CParent::EErrCode err_code,const string& message) throw()
        : CParent(file, line, prev_exception,
                 (typename CParent::EErrCode) CException::eInvalid, message)
    NCBI_EXCEPTION_DEFAULT_IMPLEMENTATION_TEMPL(CErrnoTemplException<TBase>, CParent)
};
```

The template class is derived form another template class, the ***ErrnoTemplExceptionEx*** which implements a parent class with the template parameter *TBase*. The parent ***ErrnoTemplExceptionEx*** class implements the basic exception methods such as ***ReportExtra()***, ***GetErrCode()***, ***GetErrno()***, ***GetType()***. The ***ErrnoTemplExceptionEx*** class has an ***int*** data member called `m_Errno`. The constructor automatically adds information about the most recent error state as obtained via the global system variable `errno` to this data member.

The constructor for the derived ***CErrnoTemplException*** class is defined in terms of the `NCBI_EXCEPTION_DEFAULT_IMPLEMENTATION_TEMPL` macro which defines the program code for implementing the constructor.

The *TBase* template parameter is an exception base class such as ***CException*** or ***CCoreException***, or another class similar to these. The ***CStrErrAdapt::strerror*** template parameter is a function defined in an adaptor class for getting the error description string. The ***CErrnoTemplException*** has only one error core - `eErrno` defined in the parent class, ***ErrnoTemplExceptionEx***. To analyze the actual reason of the exception one should use ***GetErrno()*** method:

```
int GetErrno(void) const;
```

The ***CErrnoTemplException*** is used to create exception classes. Here is an example of how the ***CExecException*** class is created from ***CErrnoTemplException***. In this example, the *TBase* template parameter is the exception base class ***CCoreException***:

```
/////////////////////////////////////////////////////////////////////////////
///
/// CExecException --
```

```
///
/// Define exceptions generated by CExec.
///
/// CExecException inherits its basic functionality from
/// CErrnoTemplException<CCoreException> and defines additional error codes
/// for errors generated by CExec.

class NCBI_XNCBI_EXPORT CExecException :
                        public CErrnoTemplException<CCoreException>
{
public:
    /// Error types that CExec can generate.
    enum EErrCode {
        eSystem,        ///< System error
        eSpawn          ///< Spawn error
    };

    /// Translate from the error code value to its string representation.
    virtual const char* GetErrCodeString(void) const
    {
        switch (GetErrCode()) {
        case eSystem: return "eSystem";
        case eSpawn:  return "eSpawn";
        default:      return CException::GetErrCodeString();
        }
    }

    // Standard exception boilerplate code.
    NCBI_EXCEPTION_DEFAULT(CExecException,
                           CErrnoTemplException<CCoreException>);
};
```

The CParseException Class

The ***CParseTemplException*** is a template class whose parent class is the template parameter *TBase*. The ***CParseTemplException*** class includes an additional ***int*** data member, called `m_Pos`. This class was specifically defined to support complex parsing tasks, and its constructor requires that positional information be supplied along with the description message. This makes it impossible to use the standard `NCBI_THROW` macro to throw it, so we defined two additional macros:

```
/// Throw exception with extra parameter.
///
/// Required to throw exceptions with one additional parameter
/// (e.g. positional information for CParseException).
#define NCBI_THROW2(exception_class, err_code, message, extra) \
    throw exception_class(__FILE__, __LINE__, \
        0,exception_class::err_code, (message), (extra))

/// Re-throw exception with extra parameter.
```

```
///
/// Required to re-throw exceptions with one additional parameter
/// (e.g. positional information for CParseException).
#define NCBI_RETHROW2(prev_exception,exception_class,err_code,message,extra) \
    throw exception_class(__FILE__, __LINE__, \
        &(prev_exception), exception_class::err_code, (message), (extra))
```

Macros for Standard C++ Exception Handling

The C++ ***throw()*** statement provides a mechanism for specifying the types of exceptions that may be thrown by a function. Functions that do **not** include a ***throw()*** statement in their declaration can throw any type of exception, but where the ***throw()*** statement **is** used, undeclared exception types that are thrown will cause ***std::unexpected()*** to be raised. Various compilers handle these events differently, and the first two macros listed in Table 5, (`THROWS(())`, `THROWS_NONE`, are provided to support platform-independent exception specifications.

Table 5. Platform Independent Exception Macros

Macro	C++ Equivalent	Synopsis
`THROWS((types))`	*throw(types)*	Defines the type of exceptions thrown by the given function. `types` may be a single object type or a comma delimited list.
`THROWS_NONE`	*throw()*	Specifies that the given function throws no exceptions.
`STD_CATCH(message)`	*catch(std::exception)*	Provides uniform handling of all exceptions derived from ***std::exception***.
`STD_CATCH_ALL(message)`	*catch(...)*	Applies `STD_CATCH()` to ***std::exception*** derived objects; catches non-standard exceptions and generates an "Unknown exception" message.

The `catch` macros provide uniform, routine exception handling with minimal effort from the programmer. We provide a convenient `STD_CATCH()` macro to print formatted messages to the application's *diagnostic stream*. For example, if ***F()*** throws an exception of the form:

```
throw std::runtime_error(throw-msg)
```

then

```
try {F();}
STD_CATCH(catch-msg);
```

will generate a message of the form:

```
Error: [catch-msg] Exception: throw-msg
```

In this example, the generated message starts with the Error tag, as that is the severity level for the default diagnostic stream. User-defined classes that are derived from ***std::exception*** will be treated uniformly in the same manner. The *throw* clause in this case creates a new instance of ***std::runtime_error*** whose data member `desc` is initialized to `throw-msg`. When the exception is then caught, the exception's member function ***what()*** can be used to retrieve that message.

The `STD_CATCH_ALL` macro catches all exceptions. If however, the exception caught is **not** derived from ***std::exception***, then the *catch* clause cannot assume that ***what()*** has been defined for this object, and a default message is generated:

```
Error: [catch-msg] Exception: Unknown exception
```

Exception Tracing

Knowing exactly where an exception first occurs can be very useful for debugging purposes. ***CException*** class has this functionality built in, so it is highly recommended to use exceptions derived from it. In addition to this a set of `THROW*_TRACE()` macros defined in the NCBI C++ Toolkit combine exception handling with trace mechanisms to provide such information.

The most commonly used of these macros, *THROW1_TRACE(class_name, init_arg)*, instantiates an exception object of type `class_name` using `init_arg` to initialize it. The definition of this macro is:

```
/// Throw trace.
///
/// Combines diagnostic message trace and exception throwing. First the
/// diagnostic message is printed, and then exception is thrown.
///
/// Arguments can be any exception class with  the specified initialization
/// argument. The class argument need not be derived from std::exception as
/// a new class object is constructed using the specified class name and
/// initialization argument.
///
/// Example:
/// -  THROW1_TRACE(runtime_error, "Something is weird...");
#  define THROW1_TRACE(exception_class, exception_arg) \
    throw NCBI_NS_NCBI::DbgPrint(__FILE__, __LINE__, \
        exception_class(exception_arg), #exception_class)
```

From the ***throw()*** statement here, we see that the object actually being thrown by this macro is the value returned by DbgPrint(). ***DbgPrint()*** in turn calls ***DoDbgPrint***(). The latter is an overloaded function that simply creates a diagnostic stream and writes the file name, line number, and the exception's ***what()*** message to that stream. The exception object (which is of type `class_name`) is then the value returned by ***DbgPrint()***.

More generally, three sets of `THROW*_TRACE` macros are defined:

- THROW0_TRACE(exception_object)
- THROW0p_TRACE(exception_object)
- THROW0np_TRACE(exception_object)
- THROW1_TRACE(exception_class, exception_arg)
- THROW1p_TRACE(exception_class, exception_arg)
- THROW1np_TRACE(exception_class, exception_arg)
- THROW_TRACE(exception_class, exception_args)
- THROWp_TRACE(exception_class, exception_args)
- THROWnp_TRACE(exception_class, exception_args)

The first three macros (THROW0*_TRACE) take a single argument, which may be a newly constructed exception, as in:

```
THROW0_TRACE(runtime_error("message"))
```

or simply a *printable* object to be thrown, as in:

```
THROW0_TRACE("print this message")
```

The THROW0_TRACE macro accepts either an exception object or a string as the argument to be thrown. The THROW0p_TRACE macro generalizes this functionality by accepting any *printable* object, such as *complex(1,3)*, as its single argument. Any object with a defined output operator is, of course, printable. The third macro generalizes this one step further, and accepts aggregate arguments such as ***vector<T>***, where T is a printable object. Note that in cases where the object to be thrown is not a ***std::exception***, you will need to use STD_CATCH_ALL or a custom catch to catch the thrown object.

The remaining six macros accept two arguments: an "exception" class name and an initialization argument, where both arguments are also passed to the trace message. The class argument need not actually be derived from ***std::exception***, as the pre-processor simply uses the class name to construct a new object of that type using the initialization argument. All of the THROW1*_TRACE macros assume that there is a single initialization argument. As in the first three macros, THROW1_TRACE(), THROW1p_TRACE() and THROW1np_TRACE() specialize in different types of printable objects, ranging from exceptions and numeric and character types, to aggregate and container types.

The last three macros parallel the previous two sets of macros in their specializations, and may be applied where the exception object's constructor takes multiple arguments. (See also the discussion on Exception handling).

It is also possible to specify that execution should abort immediately when an exception occurs. By default, this feature is not activated, but the ***SetThrowTraceAbort()*** function can be used to activate it. Alternatively, you can turn it on for the entire application by setting either the `$ABORT_ON_THROW` environment variable, or the application's registry *ABORT_ON_THROW* entry (in the *[DEBUG]* section) to an arbitrary non-empty value.